

CREACIÓN DE INTERFACES DE USUARIO

PRÁCTICAS 2018/19

5 de febrero de 2019

Modesto Castrillón Santana, José Daniel Hernández Sosa
Universidad de Las Palmas de Gran Canaria
Escuela de Ingeniería en Informática

Esta obra está bajo una licencia de Creative Commons Reconocimiento - No Comercial
4.0 Internacional 

Índice general

1. Introducción a Processing	5
1.1. Introducción	5
1.2. Instalación y entorno de programación	5
1.3. Programación en modo básico	7
1.3.1. Dibujo de primitivas básicas	7
1.3.2. Variables	11
1.3.3. Tipos de datos	12
1.3.4. Repeticiones	14
1.3.5. Dibujar un tablero de ajedrez	15
1.4. Programación en modo continuo	17
1.4.1. El bucle infinito de ejecución	17
1.4.2. Control	22
1.4.3. Interacción	28
1.4.3.1. Teclado	29
1.4.3.2. Ratón	31
1.4.4. Sonido	35
1.5. Otras referencias y fuentes	36
1.6. Tarea	37
2. Superficie de revolución	39
2.1. PShape	39
2.2. P3D	43
2.3. Sólido de revolución	46
2.3.1. Rotación de un punto 3D	47
2.4. Tarea	49

Práctica 1

Introducción a Processing

1.1. INTRODUCCIÓN

Processing [Processing Foundation](#) [Accedido Enero 2019a] es un proyecto de código abierto con fines creativos basado en el lenguaje Java, y concebido como un cuaderno de dibujo para estudiantes, artistas informáticos, programadores y diseñadores. La facilidad sintáctica de Java, y la enorme comunidad existente, sirven de gran apoyo, ofreciendo un conjunto de herramientas de código abierto para la creación de aplicaciones creativas. Su diseño pretende facilitar la programación que integre imágenes, animación, sonido e interacción, ofreciendo un entorno de desarrollo para prototipado rápido que además de las posibilidades de visualización, permite la integración de sensores y actuadores. Siendo además factible el desarrollo para Android, p5.js, Python, etc.

Es ampliamente utilizado en las mencionadas comunidades tanto para el aprendizaje, como la creación de prototipos y la producción audiovisual. Cubre por tanto necesidades no sólo para enseñar los fundamentos de programación, sino también como cuaderno de prototipos software, o herramienta de producción profesional. Processing está disponible en el siguiente [enlace](#)¹, si bien en la sección 1.5 se relacionan otros recursos con ejemplos, demos y bibliografía.

1.2. INSTALACIÓN Y ENTORNO DE PROGRAMACIÓN

La instalación requiere previamente realizar la descarga a través del mencionado enlace, y descomprimir. Una vez realizada la instalación, al lanzar, se presenta la interfaz, que en caso

¹<http://processing.org/>

de no estar español, puede escogerse a través del menú con *File* → *Preferences*. Es posible acceder a diversos ejemplos a través de la barra de menú *Archivo* → *Ejemplos*.

El entorno de desarrollo de Processing (PDE), ver figura 1.1, consiste en un editor de texto para escribir código, un área de mensajes, una consola de texto, fichas de gestión de archivos, una barra de herramientas con botones para las acciones comunes, y una serie de menús. Cuando se ejecuta un programa, se abre en una nueva ventana llamada la ventana de visualización (*display window*).

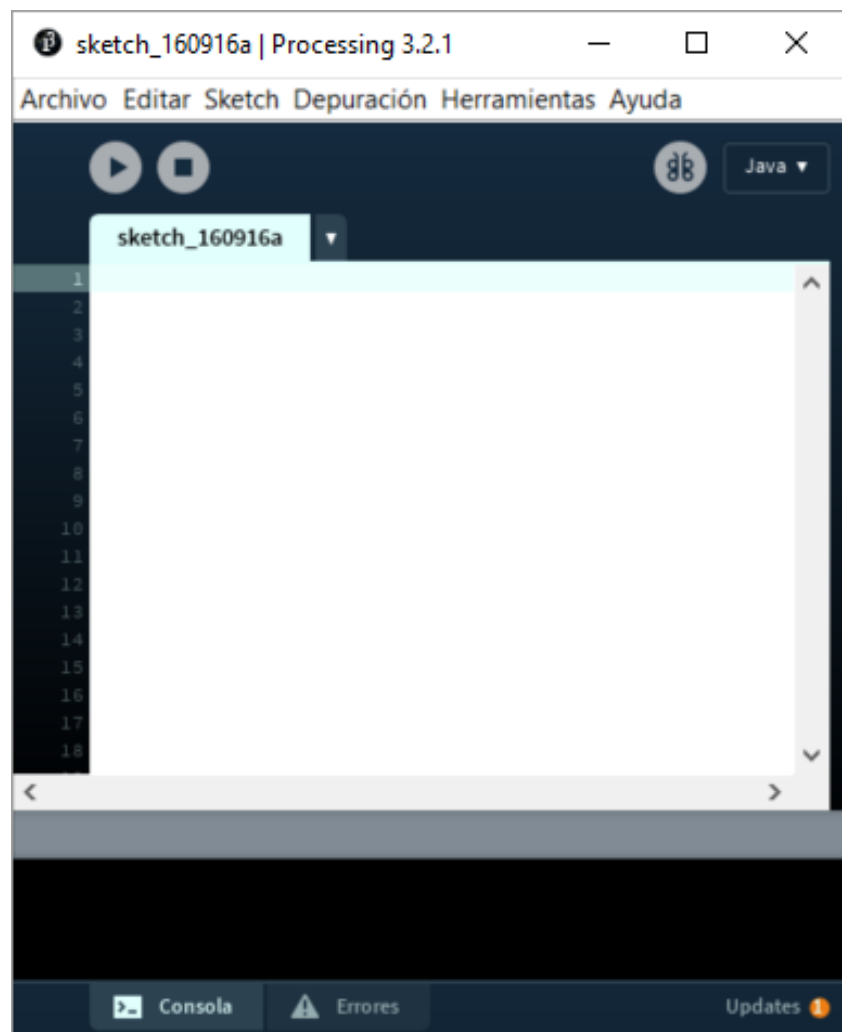


Figura 1.1: Imagen del entorno de programación, PDE, de Processing.



Cada pieza de software escrito con Processing se denomina boceto o *sketch*. Se escribe a través del editor de texto, disponiendo de las funciones típicas para cortar y pegar, así como las de búsqueda y reemplazo de texto.

El área de mensajes, en la parte inferior, ofrece información de la salida de texto del

programa en ejecución, al hacer uso de las funciones *print()* y *println()*, además de mensajes de error, tanto en ejecución como durante la edición. Las utilidades para la depuración integradas en el entorno están disponibles desde la versión 2.0b7.

Los botones de la barra de herramientas permiten ejecutar y detener programas, ver Tabla 1.1. Comandos adicionales se encuentran dentro de la barra de menú: *Archivo*, *Editar*, *Sketch*, *Depuración*, *Herramientas*, *Ayuda*. Los submenús son sensibles al contexto, lo que significa que sólo los elementos pertinentes a la labor que se está llevando a cabo están disponibles.

Cuadro 1.1: Resumen de comandos habituales

	Ejecutar	Compila el código, abre una ventana de visualización, y ejecuta el programa
	Detener	Finaliza la ejecución de un programa

1.3. PROGRAMACIÓN EN MODO BÁSICO

Processing distingue dos modos de programación: el básico, y el continuo, se presentan brevemente ambos. El modo básico permite la elaboración de imágenes estáticas, es decir que no se modifican. Líneas simples de código tienen una representación directa en la pantalla.

1.3.1. Dibujo de primitivas básicas

Un ejemplo mínimo de dibujo de una línea entre dos puntos de la pantalla se presenta en el listado 1.1.

Listado 1.1: Ejemplo de dibujo de una línea

```
line(0,0,10,10);
```

Se debe tener en cuenta que se emplea el sistema de coordenadas cartesiano, como es habitual, teniendo su origen en la esquina superior izquierda. Esto quiere decir que para cualquier dimensión de ventana, la coordenada [0,0] se corresponde con la esquina superior izquierda como se ilustra en la figura 1.2.

Processing también puede dibujar en tres dimensiones. En la superficie de la imagen, la coordenada *z* es cero, con valores *z* negativos moviéndose hacia atrás en el espacio, ver

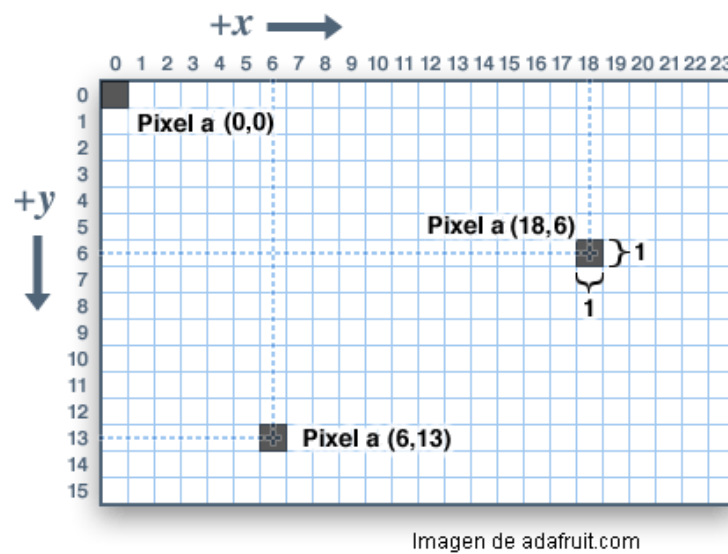


Figura 1.2: Sistema de coordenadas de la pantalla.

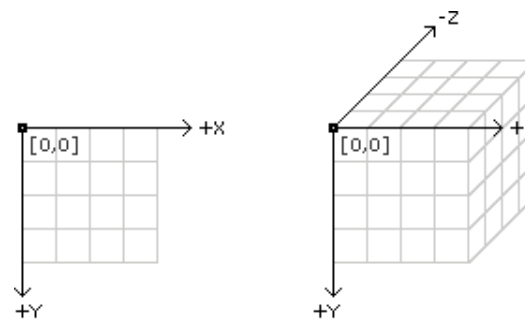


Figura 1.3: Sistema de coordenadas en 2 y 3 dimensiones (imagen de processing.org).

figura 1.3. Cuando se realiza el dibujo en 3D simulado, la cámara se coloca en el centro de la pantalla.

Un siguiente paso consiste en dibujar dos líneas, modificando el color del pincel para cada una de ellas con la función *stroke*. En el listado 1.2, se especifica el color con una tripleta RGB. A través del [enlace²](http://www.w3schools.com/colors/colors_rgb.asp) puede practicarse con el espacio de color RGB (rojo, verde y azul) modificando los valores de cada canal. RGB es el modelo de color por defecto, si bien puede adoptarse otro con *colormode*.

Listado 1.2: Dibujo de dos líneas modificando el color

```
stroke(255,0,0); // Tripletas RGB
line(0,0,10,10);

stroke(0,255,0);
line(30,10,10,30);
```

²http://www.w3schools.com/colors/colors_rgb.asp

Señalar que el comando *stroke*, puede especificar un único valor, entre 0 y 255, que se interpreta como tono de gris, p.e. *stroke(0)*; especifica el color negro, y *stroke(255)*; el blanco. Es también posible indicar la tripla RGB como un valor hexadecimal *stroke(#9ACD32)*;

Cabe destacar que es posible establecer de forma arbitraria el tamaño de la ventana de visualización con el comando *size*, como se muestra en el listado 1.3.

Listado 1.3: Una línea en una ventana de 640×360 píxeles

```
size(640, 360);  
stroke(255,0,0);  
line(0,0,10,10);
```

Los ejemplos previos fijan el color de las líneas, una posibilidad es asignarlo a partir de valores aleatorios haciendo uso de *random*, tal y como se muestra el listado 1.4. El valor entre paréntesis fija el valor máximo admisible.

Listado 1.4: Dibujo de una línea con color aleatorio

```
size(640, 360);  
stroke(random(255),random(255),random(255));  
line(0,0,10,10);
```

Una vez comprendido el dibujo de líneas, puede plantearse pintar un cuadrado haciendo uso de cuatro líneas. ¿Cómo se deducen los puntos extremos de cada segmento? Pueden ser útiles hacer papel y lápiz.

El listado 1.5 dibuja el cuadrado considerando como esquinas superior izquierda e inferior derecha respectivamente los puntos (30,30) y (60,60). El ejemplo además define un color de fondo con *background*, y un grosor de línea con *strokeWeight*.

Listado 1.5: Dibujo de un cuadrado con cuatro líneas

```
background(128);  
  
size(400,400);  
  
strokeWeight(2); //Modifica el grosor del pincel  
line(30,30,30,60);  
line(30,60,60,60);  
line(60,60,60,30);  
line(60,30,30,30);
```

Como es habitual, existen comandos que facilitan el dibujo de otras primitivas sencillas, el listado 1.6 muestra el comando *rect* para dibujar en este caso un cuadrado de 10×10 . Para conocer todas las primitivas 2D, ver 2D primitives en *Ayuda* → *Referencia*.

Listado 1.6: Dibujo de un cuadrado

```
stroke(255,255,255);  
rect(0,0,10,10);
```

El color de relleno se define con *fill()*, afectando a las primitivas a partir de ese momento, ver listado 1.7. Al igual que el resto de comandos que definen un color, la especificación de un único valor se interpreta como nivel de gris (0 negro, 255 blanco). Si se indicaran 4, el último define la transparencia, el canal alfa. Las funciones *noFill* y *noStroke* cancelan respectivamente el relleno y el borde de las primitivas.

Listado 1.7: Dibujo de una cuadrado sólido

```
stroke(255,0,255);  
fill(232,123,87);  
rect(0,0,10,10);
```

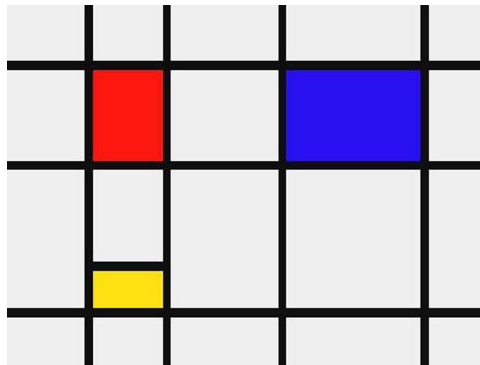
A modo de resumen, el listado 1.8 muestra el uso de varias primitivas 2D.

Listado 1.8: Primitivas 2D

```
size(450,450);  
  
stroke(128);  
fill(128);  
  
ellipse(200,300,120,120); //Por defecto modo con coordenadas del centroy di'ametro  
  
stroke(255,0,255);  
noFill();  
strokeWeight(2);  
ellipse(400,300,60,60);  
  
stroke(123, 0, 255);  
strokeWeight(10);  
ellipse(40,123,60,60);  
  
stroke(0);  
strokeWeight(1);  
line(40,123,400,300);  
  
triangle(10, 240, 50, 245, 24, 280);  
  
fill(0);  
rect(190,290,30,50);  
  
stroke(255,0,0);  
fill(255,0,0);  
bezier(5,5,10,10,310,320,320,20);
```

Con los comandos ya conocidos, sería posible componer un dibujo estático combinando varias primitivas y colores (*rect*, *ellipse*, *line*, ...). Incluso podría reproducir el Mondrian de la

figura 1.4.

**Figura 1.4:** Piet Mondrian, *Composición con rojo, amarillo y azul* (1930).

1.3.2. Variables

El uso de variables aporta posibilidades en la escritura de código. Para comenzar, utilizamos algunas de las presentes durante la ejecución, como son las dimensiones de la ventana que se almacenan respectivamente en las variables *width* y *height*. Dichas variables se utilizan en el listado 1.9, para pintar una estrella simple en el centro de la ventana, con independencia de las dimensiones fijadas con *size()*.

Listado 1.9: Dibujo de una estrella o asterisco

```
line (width/2-10,height/2-10,width/2+10,height/2+10);
line (width/2+10,height/2-10,width/2-10,height/2+10);
line (width/2,height/2-10,width/2,height/2+10);
line (width/2+10,height/2,width/2-10,height/2);
```

Cada variable es básicamente un alias o símbolo que nos permite hacer uso de una zona de almacenamiento en memoria. Dado que recordar la dirección de memoria, un valor numérico, es engorroso, se hace uso de un nombre o identificador que permite darle mayor semántica a aquello que contiene la variable. En el caso de las variables del sistema mencionadas, *width* es una variable que justamente almacena el ancho de la ventana. Las variables se caracterizan por el nombre, el valor que contienen, su dirección, y el tipo de datos.

Una gran ventaja del uso de variables es que un programador puede definir y utilizar sus propias variables a conveniencia. En Processing es necesario declarar cualquier variable antes de utilizarla. En el listado 1.10 utiliza **l** para establecer el tamaño de la estrella.

Listado 1.10: Dibujo de una estrella variable

```
int l=10;

line (width/2-1, height/2-1, width/2+1, height/2+1);
line (width/2+1, height/2-1, width/2-1, height/2+1);
line (width/2, height/2-1, width/2, height/2+1);
line (width/2+1, height/2, width/2-1, height/2);
```

El ejemplo del listado 1.11 dibuja una línea y un círculo de un determinado radio, haciendo uso de la función *ellipse* definiendo previamente el color de relleno.

Listado 1.11: Dibujo de un círculo

```
int Radio = 50 ;

size (500,500);
background(0);
stroke(80);
line (230,220,285,275);
fill (150,0,0);
ellipse (210,100,Radio, Radio);
```

1.3.3. Tipos de datos

Processing está basado en Java, por lo que deben asumirse cualquier característica de dicho lenguaje en Processing. Varios tipos de variables se muestran en el listado 1.12. Tener presente que las variables se deben declarar explícitamente y asignar las variables antes de llamar o de realizar una operación sobre ellas.

Listado 1.12: Ejemplos de tipos de variables

```
// Cadenas
String myName = "supermanoeuvre";
String mySentence = " was born in ";
String myBirthYear = "2006";

// Concatenar
String NewSentence = myName + mySentence + myBirthYear;
System.out.println(NewSentence);

// Enteros
int myInteger = 1;
int myNumber = 50000;
int myAge = -48;

// Reales
float myFloat = 9.5435;
float timeCount = 343.2;

// Booleanos // True o False
boolean mySwitch = true;
```

```
boolean mySwitch2 = false;
```

El listado 1.13 incluye ejemplos de accesos a vectores.

Listado 1.13: Uso de vectores

```
%\begin{lstlisting}[style=C++]
// Lista de Cadenas
String [] myShoppingList = new String[3];
myShoppingList[0] = "bananas";
myShoppingList[1] = "coffee ";
myShoppingList[2] = "tuxedo ";

// Lista de enteros
int [] myNumbers = new int[4];
myNumbers[0] = 498;
myNumbers[1] = 23;
myNumbers[2] = 467;
myNumbers[3] = 324;

// printamos un dato de la lista
println( myNumbers[2] );

int a = myNumbers[0] + myNumbers[3];
println( a );
```

Processing incluye la clase *ArrayList* de Java, que no requiere conocer su tamaño desde el inicio. De esta forma se facilita añadir objetos a la lista, ya que el tamaño de la lista aumenta o decrece de forma automática, ver listado 1.14.

Listado 1.14: Uso del tipo ArrayList

```
ArrayList lista = new ArrayList();
int i = 0;

while (i<4){
    lista.add(i+3);
    i=i+1;
}

println("\nLos datos son: \n");
Iterator iter = lista.iterator();
while(iter.hasNext()){
    println(iter.next());
}

ArrayList myVectorList ;
myVectorList = new ArrayList();

// Asignamos objetos
myVectorList.add( new PVector(51,25,84) );
myVectorList.add( new PVector(98,3,54) );
```

```
// o //
PVector myDirection = new PVector(98,3,54);
myVectorList.add( myDirection );

// Bucle para acceder a objetos usando ArrayList.size() y ArrayList.get()
for(int i = 0; i < myVectorList.size(); i++){
PVector V = (PVector) myVectorList.get(i); // ojo con el cast (PVector)
println(V);
}
```

1.3.4. Repeticiones

El código del listado 1.15 crea una ventana de dimensiones 800×800 en la que pintamos líneas verticales de arriba a abajo separadas entre ellas 100 píxeles. Recordar que la coordenada y de la parte superior es 0, y la inferior es 800 o *height* si usamos la variable correspondiente.

Listado 1.15: Dibujo de varias líneas verticales

```
size(800,800);
background(0);
stroke(255);
line(100,1,100,height);
line(200,1,200,height);
line(300,1,300,height);
line(400,1,400,height);
line(500,1,500,height);
line(600,1,600,height);
line(700,1,700,height);
```

Claramente las llamadas a la función *line* son todas muy similares, sólo varían las coordenadas x de los dos puntos. Los lenguajes de programación facilitan la especificación de llamadas repetidas por medio del uso de bucles. Una versión más compacta del dibujo de las líneas verticales se muestra en el listado 1.16. El bucle define una variable, i , a la que asignamos un valor inicial 100, un valor final, 700, y la forma en que se va modificando i con cada ejecución, en este caso añadiendo 100.

Listado 1.16: Dibujo de varias líneas verticales con un bucle

```
size(800,800);
background(0);
stroke(255);
for (int i=100;i<=700;i=i+100){
line(i,1,i,height);
}
```

Las sentencias repetitivas son particularmente útiles cuando las repeticiones son cientos o miles.

1.3.5. Dibujar un tablero de ajedrez

Esta sección aborda el dibujo de un tablero de ajedrez, que contiene 64 casillas, como muestra la figura 1.5. El listado 1.17 Fija el fondo a blanco, dibujando los cuatro recuadros negros de la primera fila del tabler.

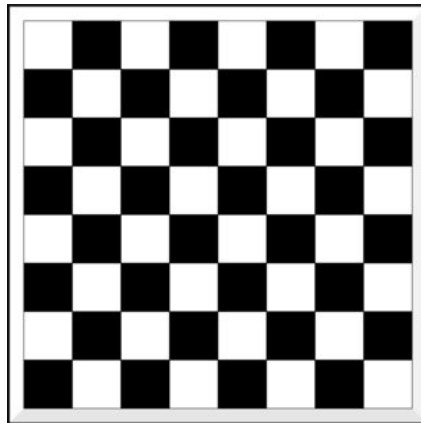


Figura 1.5: Rejilla estilo tablero de ajedrez.

Listado 1.17: Dibujo de una rejilla

```
size(800,800);
background(255);
fill(0);
//Primera fila
rect(0,0,100,100);
rect(200,0,100,100);
rect(400,0,100,100);
rect(600,0,100,100);
```

El listado 1.18 aplica un bucle *for* para pintar esas cuatro casillas negras.

Listado 1.18: Dibujo de una rejilla con un bucle

```
size(800,800);
background(255);
fill(0);
//Primera fila

for (int i=0;i<=600;i=i+200){
    rect(i,0,100,100);
}
```

El listado 1.19 da el salto para dibujar cuatro filas, cada una con su bucle particular.

Listado 1.19: Dibujo de una rejilla con varios bucles

```
size(800,800);
background(255);
```

```
fill(0);
//Primera fila

for (int i=0;i<=600;i=i+200){
    rect(i,0,100,100);
}
for (int i=0;i<=600;i=i+200){
    rect(i,200,100,100);
}
for (int i=0;i<=600;i=i+200){
    rect(i,400,100,100);
}
for (int i=0;i<=600;i=i+200){
    rect(i,600,100,100);
}
```

Como realmente cada bucle es similar a los demás, el listado 1.20 anida bucles, quedando más compacto.

Listado 1.20: Dibujo de una rejilla de ajedrez

```
size(800,800);
background(255);
fill(0);
//Primera fila

for (int j=0;j<=600;j=j+200){
    for (int i=0;i<=600;i=i+200){
        rect(i,j,100,100);
    }
}
```

Restan las otras cuatro filas, que resultan de una leve variación ya que se alternan los tonos blancos y negros. como muestra el listado 1.21.

Listado 1.21: Dibujo de un tablero de ajedrez

```
size(800,800);
background(255);
fill(0);
//Primera fila

for (int j=0;j<=600;j=j+200){
    for (int i=0;i<=600;i=i+200){
        rect(i,j,100,100);
        rect(i+100,j+100,100,100);
    }
}
```

A lo largo de este apartado, se han mostrado las posibilidades del modo básico para componer una imagen estática, en la que es posible modificar los argumentos de las llamadas,

eliminar comandos, o añadir otros, pero el resultado es siempre estático, no hay movimiento, ver listado 1.22.

Listado 1.22: Modo básico

```
int Radio = 50;
void setup()
{
  size(500,500);
  background(0);
  stroke(80);
  line(230, 220, 285, 275);
  fill(150,0,0);
  ellipse(210, 100, Radio, Radio);
}
```

1.4. PROGRAMACIÓN EN MODO CONTINUO

1.4.1. El bucle infinito de ejecución

El modo continuo, cuenta con dos métodos básicos:

- *setup()* se ejecuta una única vez al lanzar el programa
- *draw()* se ejecuta por defecto de forma continua, permitiendo la escritura de funciones personalizadas, y hacer uso de la interacción.

El ejemplo el listado 1.23 usa ambos métodos, delimitando con llaves las instrucciones asociadas a cada uno de ellos. En concreto el método *setup* realiza la inicialización, en este caso fija las dimensiones de la ventana. Por otro lado, el método *draw* se encarga de dibujar. En este ejemplo, *draw* pinta una línea con color y posición aleatoria. La gran diferencia del modo continuo es que las instrucciones contenidas en el método *draw* no se ejecutan una única vez, sino que se *llama* a dicho método de forma reiterada cada segundo, por defecto 60.

Listado 1.23: Ejemplo de dibujo de líneas de colores y posición aleatoria

```
void setup()
{
  size(400, 400);
}

void draw()
{
  stroke(random(255),random(255),random(255));
}
```

```
line(random(width),random(height),random(width),random(height));  
}
```

Haciendo uso únicamente del método *setup*, el resultado es equivalente al modo básico, dado que dicho método se ejecuta una única vez, ver listado 1.24.

Listado 1.24: Ejemplo básico

```
void setup()  
{  
  size(240,240); // Dimensiones del lienzo  
  background(128,128,128); // Color de lienzo en formato RGB  
  noStroke(); // Sin borde para las figuras  
  fill(0); // Color de relleno de las figuras (0 es negro)  
  rect(100, 100, 30, 30); // Esquina superior izquierda, ancho y alto  
}
```

El código del listado 1.25 dibuja cuatro círculos en la pantalla y utiliza además una función propia llamada *circles()*. Observa el modo en que se define, y el uso de las llaves para delimitar las instrucciones contenidas en la función. En este caso concreto, el código de *draw()* sólo se ejecuta una vez, porque en *setup()* se llama a la función *noLoop()*, que cancela la repetición, resultando equivalente al modo básico.

Listado 1.25: Ejemplo de uso del método *draw*

```
void setup() {  
  size(200, 200);  
  noStroke();  
  background(255);  
  fill(0, 102, 153, 204);  
  smooth();  
  noLoop();  
}  
  
void draw() {  
  circles(40, 80);  
  circles(90, 70);  
}  
  
void circles(int x, int y) {  
  ellipse(x, y, 50, 50);  
  ellipse(x+20, y+20, 60, 60);  
}
```

En general la ejecución reiterada tiene sentido cuando exista un cambio en aquello que dibujamos ya sea por movimiento, modificación del color, etc. El ejemplo del listado 1.26 dibuja líneas desde un punto fijo, con el otro extremo aleatorio, variando también el color de forma aleatoria.

Listado 1.26: Dibujo de líneas desde un punto

```
void setup() {  
  size(400, 400);  
  background(0);  
}  
  
void draw() {  
  stroke(0,random(255),0);  
  line(50, 50, random(400), random(400));  
}
```

En el listado 1.27 se fuerza a que sean líneas aleatorias, pero verticales y blancas.

Listado 1.27: Dibujo de líneas blancas verticales

```
void setup() {  
  size(400, 400);  
}  
  
void draw() {  
  stroke(255);  
  
  float dist_izq=random(400);  
  line(dist_izq, 0, dist_izq, 399);  
}  
  
void setup() {  
  size(400, 400);  
}  
  
void draw() {  
  stroke(random(200,256),random(200,256),random(50,100)); //entre dos valores  
  
  float dist_izq=random(400);  
  line(dist_izq, 0, dist_izq, 399);  
}
```

Como se comentaba anteriormente, el color de fondo de la ventana se fija con la llamada al método *background*. Este comando nos puede ser útil si queremos forzar que se borre la pantalla antes de dibujar de nuevo, ver listado 1.28.

Listado 1.28: Ejemplo de dibujo con borrado

```
void setup()  
{  
  size(400, 400);  
}  
  
void draw()  
{  
  background(51); //Borra cada vez antes de pintar  
  stroke(random(255),random(255),random(255));  
  line(random(width),random(height),random(width),random(height));  
}
```

```
}
```

El ejemplo del listado 1.29 introduce el uso del método *frameRate*, que fija el número de llamadas al método *draw* por segundo. En este caso dibuja líneas aleatorias, pero observa la tasa de refresco.

Listado 1.29: Ejemplo de dibujo de líneas a distinta frecuencia

```
void setup() {  
  size(400, 400);  
  frameRate(4);  
}  
  
void draw() {  
  background(51);  
  
  line(0, random(height), 90, random(height));  
}
```

La tasa de refresco puede además controlarse a través de una serie de funciones:

- *loop()*: Reestablece las llamadas a *draw()*, es decir el modo continuo.
- *noLoop()*: Detiene el modo continuo, no se realizan llamadas a *draw()*.
- *redraw()*: Llama a *draw()* una sólo vez.

El código del listado 1.30 varía la tasa de refresco en base al número de ejecuciones y el evento de pulsado de un botón del ratón.

Listado 1.30: Controlando la tasa de refresco

```
int frame = 0;  
void setup() {  
  size(100, 100);  
  frameRate(30);  
}  
void draw()  
{  
  if (frame > 60)  
  { // Si han pasado 60 ejecuciones (frames) desde el comienzo  
    noLoop(); // para el programa  
    background(0); // y lo pone todo a negro.  
  }  
  else  
  { // En otro caso, pone el fondo  
    background(204); // a un gris claro  
    line(mouseX, 0, mouseX, 100); // dibuja  
    line(0, mouseY, 100, mouseY);  
    frame++;  
  }  
}
```

```
}  
}  
void mousePressed() {  
    loop();  
    frame = 0;  
}
```

El listado 1.31 ilustra un ejemplo de llamada a *redraw*.

Listado 1.31: Ejemplo con redraw

```
void setup()  
{  
    size(100, 100);  
}  
void draw() {  
    background(204);  
    line(mouseX, 0, mouseX, 100);  
}  
void mousePressed() {  
    redraw(); // Llama a draw() una vez  
}
```

Como se ha comentado anteriormente, por defecto se establece el modo continuo con una frecuencia de 60 llamadas por segundo al método *draw()*- El código del listado 1.32 provoca el desplazamiento de un círculo por la ventana.

Listado 1.32: Desplazamiento del círculo

```
int Radio = 50;  
int cuenta = 0;  
  
void setup()  
{  
    size(500,500);  
    background(0);  
}  
  
void draw()  
{  
    background(0);  
    stroke(175,90);  
    fill(175,40);  
    println("Iteraciones: " + cuenta);  
    ellipse(20+cuenta, height/2, Radio, Radio);  
    cuenta++;  
}
```

Finalizamos el apartado señalando que además de dibujar elementos geométricos, el listado 1.33 muestra que es posible escribir.

Listado 1.33: Hola mundo

```
void setup()
{
  background(128,128,128); // Color de lienzo en formato RGB
  noStroke();             // Sin borde para las figuras
  fill(0);                // Color de relleno de las figuras (0 es negro)

  // Carga una fuente en particular
  textFont(createFont("Georgia",24));
  textAlign(CENTER, CENTER);

  // Escribe en la ventana
  text("¡Hola mundo!", 100,100);
}
```

1.4.2. Control

En el ejemplo del listado 1.32 se desplaza un círculo por la ventana haciendo uso de una variable. El listado 1.34 presenta una variante algo más reducida.

Listado 1.34: Manejo de variables para dibujar circunferencias

```
int cir_x=0;

void setup() {
  size(400, 400);
  noStroke();
  fill(100,255,32); //Color de relleno
}

void draw() {
  background(127);
  ellipse(cir_x,50,50,50);

  cir_x=cir_x+1;//Modifica la coordenada x del centro de la figura
}
```

La modificación mostrada en el listado 1.35 evita que desaparezca la figura a la derecha de la ventana. Conociendo las dimensiones de la ventana, controlamos el valor de la posición, reiniciando el valor de la coordenada **x** del círculo cuando llega al borde derecho.

Listado 1.35: Controlando el regreso de la figura

```
int cir_x=0;

void setup() {
  size(400, 400);
  noStroke();
  fill(100,255,32);
}
```

```
void draw() {  
    background(127);  
  
    ellipse(cir_x,50,50,50);  
  
    cir_x=cir_x+1;  
    if (cir_x>=width)  
    {  
        cir_x=0;  
    }  
}
```

El listado 1.36 integra dos círculos a distintas velocidades.

Listado 1.36: Con dos círculos

```
float cir_rap_x = 0;  
float cir_len_x = 0;  
  
void setup() {  
    size(400,400);  
    noStroke();  
}  
  
void draw() {  
    background(27,177,245);  
  
    fill(193,255,62);  
    ellipse(cir_len_x,50, 50, 50);  
    cir_len_x = cir_len_x + 1;  
  
    fill(255,72,0);  
    ellipse(cir_rap_x,50, 50, 50);  
    cir_rap_x = cir_rap_x + 5;  
  
    if(cir_len_x > 400) {  
        cir_len_x = 0;  
    }  
    if(cir_rap_x > 400) {  
        cir_rap_x = 0;  
    }  
}
```

El listado 1.37 modifica la forma del círculo más lento de forma aleatoria cuando se dan ciertas circunstancias.

Listado 1.37: Simulando un latido

```
float cir_rap_x = 0;  
float cir_len_x = 0;  
  
void setup() {  
    size(400,400);
```

```
noStroke();
}

void draw() {
  background(27,177,245);

  float cir_len_tam = 50;

  if(random(10) > 9) {
    cir_len_tam = 60;
  }

  fill(193,255,62);
  ellipse(cir_len_x,50, cir_len_tam, cir_len_tam);
  cir_len_x = cir_len_x + 1;

  fill(255,72,0);
  ellipse(cir_rap_x,50, 50, 50);
  cir_rap_x = cir_rap_x + 5;

  if(cir_len_x > 400) {
    cir_len_x = 0;
  }
  if(cir_rap_x > 400) {
    cir_rap_x = 0;
  }
}
```

Se aprecia cierto efecto de pausa, se debe a que la condición no tiene en cuenta el radio para hacer reaparecer la figura. En el listado 1.38 lo modifica con una aparición más ajustada.

Listado 1.38: Aparición ajustada

```
int cir_x=0;

void setup() {
  size(400, 400);

  noStroke();

  fill(100,255,32);
}

void draw() {
  background(127);

  ellipse(cir_x,50,50,50);

  cir_x=cir_x+1;

  if (cir_x-50/2>width)
  {
    cir_x=25;
```



```
}  
  
if ( cir_x+50/2>=width)  
{  
    ellipse ( cir_x-width,50,50,50);  
}  
}
```

En los ejemplos previos, la modificación de la coordenada x es siempre un incremento. El listado 1.39 produce un efecto de rebote al llegar al extremo, modificando el signo del movimiento, para que pueda ser incremento o decremento. De esta forma se consigue que el círculo rebote en ambos lados.

Listado 1.39: Provocando el rebote

```
int pos=0;  
int mov=1;  
  
void setup() {  
    size(400,400);  
}  
  
void draw() {  
    background(128);  
    ellipse(pos,50,30,30);  
    pos=pos+mov;  
  
    if (pos>=400 || pos<=0)  
    {  
        mov=-mov;  
    }  
}
```

El listado 1.40 integra un *jugador*, que se desplaza en vertical acompañando al ratón, y también puede alterar el movimiento del círculo cuando haya choque.

Listado 1.40: Frontón

```
int posX=0;  
int posY=50;  
int D=30;  
int ancho=20;  
int alto=50;  
int mov=5;  
  
void setup() {  
    size(400,400);  
}  
  
void draw() {  
    background(128);  
    ellipse(posX,posY,D,D);
```

```
//Donde se encuentra el jugador
int jugx=width-50;
int jugy=mouseY-30;
rect(jugx, jugy, ancho, alto);

posX=posX+mov;
//verificando si hay choque
if (posX>=400 || posX<=0 || (mov>0 && jugy<=posY+D/2 && posY-D/2<=jugy+alto && jugx<=posX+D/2 && posX-D/2<=jugx+ancho))
{
    mov=-mov;
}
}
```

Añadimos un marcador contabilizando el número de veces que no controlamos la *pelota*. En el listado 1.41 introducimos una variable contador, inicialmente a cero, que modifica su valor cada vez que haya un choque con la pared de la derecha.

Listado 1.41: Integrando el marcador

```
int posX=0;
int posY=50;
int D=30;
int ancho=20;
int alto=50;
int mov=5;
int goles=0;

void setup() {
    size(400,400);
}

void draw() {
    background(128);
    ellipse(posX, posY, D, D);

    //Donde se encuentra el jugador
    int jugx=width-50;
    int jugy=mouseY-30;
    rect(jugx, jugy, ancho, alto);

    posX=posX+mov;
    //verificando si hay choque
    if (posX>=400 || posX<=0 || (mov>0 && jugy<=posY+D/2 && posY-D/2<=jugy+alto && jugx<=posX+D/2 && posX-D/2<=jugx+ancho))
    {
        mov=-mov;
        //Si choca con la derecha, es gol
        if (posX>=400)
        {
            goles=goles+1;
        }
    }
}
```

```

    }
    text("Goles "+goles , width/2-30, 20);
}

```

Sería también posible mostrar un mensaje cada vez que haya un gol, es decir, cuando se toque la pared derecha. Mostrarlo únicamente al modificar el tanteo, hará que casi no se vea, es por ello que lo hacemos creando un contador que se decrementa cada vez que se muestra el mensaje, ver listado 1.42.

Listado 1.42: Celebrando el gol

```

int posX=0;
int posY=50;
int D=30;
int ancho=20;
int alto=50;
int mov=5;
int goles=0;
int muestragol=0;

void setup() {
    size(400,400);
}

void draw() {
    background(128);
    ellipse(posX,posY,D,D);

    //Donde se encuentra el jugador
    int jugx=width-50;
    int jugy=mouseY-30;
    rect(jugx, jugy, ancho, alto);

    posX=posX+mov;
    //verificando si hay choque
    if (posX>=400 || posX<=0 || (mov>0 && jugy<=posY+D/2 && posY-D/2<=jugy+alto && jugx<=posX+D/2 && posX-D/2<=jugx+ancho))
    {
        mov=-mov;
        //Si choca con la derecha, es gol
        if (posX>=400)
        {
            goles=goles+1;
            muestragol=40;
        }
    }
    text("Goles "+goles , width/2-30, 20);
    if (muestragol>0)
    {
        text("GOOOOL", width/2-30, height-50);
        muestragol=muestragol-1;
    }
}

```

Introducir el movimiento en los dos ejes requiere modificar la coordenada y de la pelota. El código del listado 1.43 lo integra, manteniendo el rebote en las paredes, ahora también inferior y superior.

Listado 1.43: Rebote de la bola

```
float cir_x = 300;
float cir_y = 20;
// desplazamiento
float mov_x = 2;
float mov_y = -2;

void setup() {
  size(400, 200);
  stroke(214,13,255);
  strokeWeight(7);
}

void draw() {
  background(33,234,115);
  ellipse(cir_x, cir_y, 40, 40);
  cir_x = cir_x + mov_x;
  cir_y = cir_y + mov_y;

  if(cir_x > width) {
    cir_x = width;
    mov_x = -mov_x;
    println("derecha");
  }
  if(cir_y > height) {
    cir_y = height;
    mov_y = -mov_y;
    println("abajo");
  }
  if(cir_x < 0) {
    cir_x = 0;
    mov_x = -mov_x;
    println("izquierda");
  }
  if(cir_y < 0) {
    cir_y = 0;
    mov_y = -mov_y;
    println("arriba");
  }
}
```

1.4.3. Interacción

En el modo continuo será frecuente la interacción con los usuario vía teclado o ratón.

1.4.3.1. Teclado

Las acciones de teclado puede utilizarse como evento en sí o para recoger los detalles de la tecla pulsada. Por ejemplo la variable booleana *keyPressed* se activa si una tecla es presionada, en cualquier otro caso es falsa. Además, dicha variable estará activa mientras mantengamos la tecla permanezca. El código del listado 1.44 hace uso de dicha variable para desplazar una línea.

Listado 1.44: Ejemplo básico

```
int x = 20;
void setup() {
  size(100, 100);
  smooth();
  strokeWeight(4);
}
void draw() {
  background(204);
  if (keyPressed == true)
  {
    x++; // incrementamos x
  }
  line(x, 20, x+60, 80);
}
```

Es posible recuperar la tecla pulsada. La variable *key* es de tipo char y almacena el valor de la tecla que ha sido presionada recientemente. En el listado 1.45 se muestra la tecla pulsada.

Listado 1.45: Muestra la tecla

```
void setup() {
  size(100, 100);
}

void draw() {
  background(0);
  text(key, 28, 75);
}
```

Cada letra tiene un valor numérico en la Tabla ASCII, que también podemos mostrar, tal y como se realiza en el listado 1.46.

Listado 1.46: El código ASCII de la tecla

```
void setup() {
  size(100, 100);
}

void draw() {
  background(200);
  if (keyPressed == true)
```

```
{
  int x = key;
  text(key+" ASCII "+x, 20, 20 );
}
}
```

Una particularidad interesante es poder identificar las teclas especiales como por ejemplo: flechas, *Shift*, *Backspace*, tabulador y otras más. Para ello, lo primero que debemos hacer es comprobar si se trata de una de estas teclas comprobando el valor de la variable *key* == *CODED*. El listado 1.47 utiliza las teclas de las flechas cambie la posición de una figura dentro del lienzo.

Listado 1.47: Ejemplo básico

```
color y = 35;
void setup() {
  size(100, 100);
}
void draw() {
  background(204);
  line(10, 50, 90, 50);
  if (key == CODED)
  {
    if (keyCode == UP)
    {
      y = 20;
    }
    else
      if (keyCode == DOWN)
      {
        y = 50;
      }
  }
  else
  {
    y = 35;
  }
  rect(25, y, 50, 30);
}
```

También es posible detectar eventos individualizados, es decir el pulsado de una tecla en concreto. El código del listado 1.48 realiza una acción al pulsar la tecla t.

Listado 1.48: Ejemplo básico evento de teclado

```
boolean drawT = false;

void setup() {
  size(100, 100);
  noStroke();
}
```

```
void draw() {
  background(204);
  if (drawT == true)
  {
    rect(20, 20, 60, 20);
    rect(39, 40, 22, 45);
  }
}

void keyPressed()
{
  if ((key == 'T') || (key == 't'))
  {
    drawT = true;
  }
}

void keyReleased() {
  drawT = false;
}
```

1.4.3.2. Ratón

En ejemplos previos hemos visto que es posible acceder a las coordenadas del ratón, simplemente haciendo uso de las variables adecuadas desde el método *draw()* como en el código de listado 1.49.

Listado 1.49: Ejemplo de uso de las coordenadas del ratón

```
void setup() {
  size(200, 200);
  rectMode(CENTER);
  noStroke();
  fill(0, 102, 153, 204);
}

void draw() {
  background(255);
  rect(width-mouseX, height-mouseY, 50, 50);
  rect(mouseX, mouseY, 50, 50);
}
```

Es posible usar el evento, es decir, la función que asociada con el mismo, como en el listado 1.50 que modifica el color del fondo.

Listado 1.50: Ejemplo de cambio del tono de fondo

```
float gray = 0;
void setup() {
  size(100, 100);
}

void draw() {
```

```
background(gray);  
}  
void mousePressed() {  
    gray += 20;  
}
```

El ejemplode listado 1.51 emplea el evento de pulsado para pasar del modo básico al continuo.

Listado 1.51: Paso a modo continuo

```
void setup()  
{  
    size(200, 200);  
    stroke(255);  
    noLoop();  
}  
  
float y = 100;  
void draw()  
{  
    background(0);  
    line(0, y, width, y);  
    line(0, y+1, width, y+1);  
    y = y - 1;  
    if (y < 0) { y = height; }  
}  
  
void mousePressed()  
{  
    loop();  
}
```

Como puede verse, *mousePressed()* es una función enlazada a la acción de pulsar un botón del ratón. Siendo posible controlar también cuando se suelta, se mueve, o si se arrastra el ratón con un botón pulsado, ver listado 1.52.

Listado 1.52: Ejemplo evento de arrastre

```
int dragX, dragY, moveX, moveY;  
void setup() {  
    size(100, 100);  
    smooth();  
    noStroke();  
}  
void draw() {  
    background(204);  
    fill(0);  
    ellipse(dragX, dragY, 33, 33); // C\`irculo negro  
    fill(153);  
    ellipse(moveX, moveY, 33, 33); // C\`irculo gris  
}  
void mouseMoved() { // Mueve el gris
```



```
    moveX = mouseX;  
    moveY = mouseY;  
}  
void mouseDragged() { // Mueve el negro  
    dragX = mouseX;  
    dragY = mouseY;  
}
```

Como ya se ha mencionado anteriormente, las coordenadas de ratón se almacenan en las variables *mouseX* y *mouseY*, ver listado 1.53.

Listado 1.53: Coordenadas del ratón

```
void setup() {  
    size(640, 360);  
    noStroke();  
}  
  
void draw() {  
    background(51);  
    ellipse(mouseX, mouseY, 66, 66);  
}
```

Ambas son variables conteniendo las posiciones actuales del ratón, las previas están disponibles en *pmouseX* y *pmouseY*. Podemos utilizar las coordenadas del ratón para pintar a mano alzada, ver listado 1.54.

Listado 1.54: Pintado con el «pincel»

```
void setup() {  
    size(400,400);  
  
    background(128);  
}  
  
void draw() {  
    point(mouseX, mouseY);  
}
```

Es posible restringir el pintado a sólo cuando se pulse un botón del ratón, empleando una estructura condicional, ver listado 1.55.

Listado 1.55: Pintado cuando se pulsa

```
void setup() {  
    size(400,400);  
  
    background(128);  
}  
  
void draw() {
```

```
    if (mousePressed == true) {  
        point(mouseX,mouseY);  
    }  
}
```

El listado 1.56 utiliza el teclado, en concreto las teclas del cursor arriba y abajo, para cambiar grosor del pincel (y pintamos círculo), y cualquier otra tecla para alterar el color. Se identifica primero si se ha pulsado una tecla, y luego la tecla en concreto pulsada.

Listado 1.56: Grosor y color del pincel con las teclas

```
int grosor=1;  
int R=0,G=0,B=0;  
  
void setup() {  
    size(400,400);  
    background(128);  
}  
  
void draw() {  
  
    if (mousePressed == true) {  
        point(mouseX,mouseY);  
    }  
  
    if (keyPressed == true) {  
        if (keyCode == UP) {  
            grosor = grosor+1;  
            strokeWeight(grosor);  
        }  
        else  
        {  
            if (keyCode == DOWN) {  
                if (grosor>1){  
                    grosor = grosor-1;  
                    strokeWeight(grosor);  
                }  
            }  
            else  
            {  
                R=(int)random(255);  
                G=(int)random(255);  
                B=(int)random(255);  
            }  
        }  
    }  
}  
  
//Muestra del pincel  
noStroke();  
fill(128);  
rect(4,4,grosor+2,grosor+2);  
fill(R,G,B);  
ellipse(5+grosor/2,5+grosor/2,grosor,grosor);
```

```
stroke(R,G,B);  
}
```

En el listado 1.57 el radio del pincel dependa del valor de x o y del ratón.

Listado 1.57: Radio del pincel dependiente de ratón

```
void setup() {  
  size(640, 360);  
  noStroke();  
  rectMode(CENTER);  
}  
  
void draw() {  
  background(51);  
  fill(255, 204);  
  rect(mouseX, height/2, mouseY/2+10, mouseY/2+10);  
  fill(255, 204);  
  int inversaX = width-mouseX;  
  int inversaY = height-mouseY;  
  rect(inversaX, height/2, (inversaY/2)+10, (inversaY/2)+10);  
}
```

1.4.4. Sonido

De cara a la tarea, señalar que es posible realizar la reproducción de sonidos presentes en disco como se muestre en el listado 1.58. Una posible fuente de archivos wavs es el [enlace](http://freewavesamples.com/)³. Tener en cuenta que será necesario instalar la biblioteca de sonido de la *Processing Foundation* a través del menú *Herramientas->Añadir herramientas* y buscando *Sound* en la pestaña *Libraries*.

Listado 1.58: Reproduciendo sonido

```
import processing.sound.*;  
  
int pos=0;  
int mov=5;  
  
SoundFile sonido;  
  
void setup() {  
  size(400,400);  
  
  sonido = new SoundFile(this,"E:/Intro Programacion con Processing Experto/Bass-Drum-1.wav");  
}  
  
void draw() {
```

³<http://freewavesamples.com/>

```

background(128);
ellipse(pos,30,30,30);

pos=pos+mov;

if (pos>=400 || pos<=0){
    mov=-mov;
    sonido.play ( ) ;
}
}

```

Al trabajar en modo continuo, pueden presentarse efectos extraños al reproducir sonido dada su duración, a menos que se lance el sonido a través de un hilo con el método *thread*, tal y como se muestra en el listado 1.59.

Listado 1.59: Latencia del sonido

```

import processing.sound.*;

int pos=0;
int mov=5;

SoundFile sonido;

void setup() {
    size(400,400);

    sonido = new SoundFile(this,"E:/Intro Programacion con Processing Experto/Bass-Drum-1.wav");
}

void draw() {
    background(128);
    ellipse(pos,30,30,30);

    pos=pos+mov;

    if (pos>=400 || pos<=0){
        mov=-mov;
        thread ("Suenar");
    }
}

void Suenar ( ) {
    sonido.play ( ) ;
}

```

1.5. OTRAS REFERENCIAS Y FUENTES

Señalar que a través del entorno de desarrollo de Processing están disponibles un nutrido número de ejemplos, accesibles seleccionando en la barra de menú Archivo → Ejemplos,

con lo que aparece un desplegable donde por ejemplo puedes descender *Topics* → *Motion* → *Bounce*, *Basics* → *Input* → *Storinginput*. *Fun programming* propone una introducción a la programación con Processing Pazos [Accedido Enero 2019]. Otra interesante fuente de códigos ejemplo es *Open Processing* @openprocessing [Accedido Enero 2019].

Galerías de desarrollos realizados con Processing están disponibles en la propia página a través del enlace Processing exhibition archives Processing Foundation [Accedido Enero 2019b]. Un foro más amplio que describe propuestas desarrolladas no únicamente con Processing es *Creative Applications Network* CreativeApplications.Net [Accedido Enero 2019].

Bibliografía adicional sobre Processing, desde los primeros libros de la década pasada Greenberg [2007], Reas and Fry [2007], hasta obras más recientes Noble [2012], Runberg [2015]. El libro de Runberg cubre también openFrameworks y Arduino, siendo accesible desde la universidad a través del enlace.

1.6. TAREA

Realizar de forma individual, un juego similar al Pong para dos jugadores, ver figura 1.6. La propuesta realizada debe incluir rebote, marcador, sonido, movimiento inicial aleatorio, etc.

La entrega se debe realizar a través del campus virtual, remitiendo una memoria o el enlace a un screencast, que además de describir las decisiones adoptadas para la solución propuesta, identifique al estudiante, además de las referencias y herramientas utilizadas; y que cuidando el formato y estilo, debe incluir enlace al código desarrollado (p.e. github).

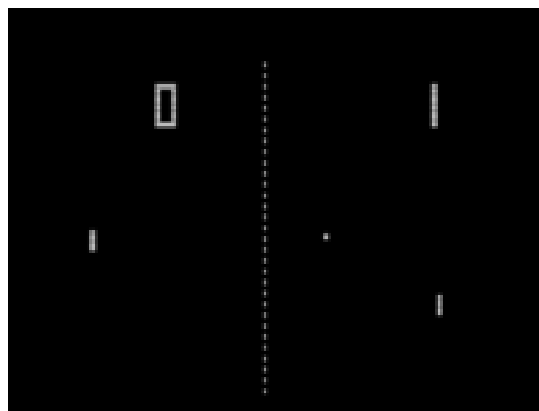


Figura 1.6: Interfaz clásica de Pong⁴.

BIBLIOGRAFÍA

CreativeApplications.Net. Creative applications network, Accedido Enero 2019. URL <http://www.creativeapplications.net/>.

Ira Greenberg. *Processing. Creative Coding and computational art*. friendof, 2007.

Joshua Noble. *Programming Interactivity*. O'Reilly, 2012. URL <http://shop.oreilly.com/product/0636920021735.do>.

@openprocessing. Openprocessing, Accedido Enero 2019. URL <https://www.openprocessing.org/>.

Abe Pazos. Fun programming, Accedido Enero 2019. URL <http://funprogramming.org/>.

Processing Foundation. Processing, Accedido Enero 2019a. URL <http://processing.org/reference/>.

Processing Foundation. Processing exhibition archives, Accedido Enero 2019b. URL <https://processing.org/exhibition/>.

Casey Reas and Ben Fry. *Processing: a programming handbook for visual designers and artists*. MIT Press, 2007. URL <https://processing.org/img/learning/Processing-Sample-070607.pdf>.

Derek Runberg. *The sparkfun guide to processing*. Sparkfun, 2015.

Práctica 2

Superficie de revolución

2.1. PSHAPE

La práctica anterior describe las primitivas básicas 2D para el dibujo de objetos tales como rectángulos y elipses. Una opción más avanzada para la creación de formas arbitrarias es hacer uso de variables *PShape* [Processing Foundation \[Accedido Enero 2019b\]](#), que por otro lado permiten acelerar el proceso de dibujado, en particular cuando los objetos crecen en complejidad.

Listado 2.1: Dibujando un rectángulo con *rect*

```
void setup() {  
  size(400, 400);  
}  
void draw() {  
  background(50);  
  stroke(255);  
  fill(127);  
  rect(mouseX, mouseY, 100, 50);  
}
```

El listado 2.1 sigue las pautas de la práctica anterior, haciendo uso de *rect* para dibujar un rectángulo que acompaña al movimiento del puntero sobre la ventana. Como ejemplo ilustrativo de una variable *PShape*, el listado 2.2 muestra el código para obtener el mismo resultado sin necesidad de la función *rect*, si bien introduciendo la traslación 2D con la función *translate*.

Listado 2.2: Dibujando un rectángulo como variable *PShape*

```
PShape rectangle;  
  
void setup() {  
  size(400, 400, P2D);
```

```
//La forma
rectangle = createShape(RECT,-50,-25,100,50);
//Aspectos de dibujo
rectangle.setStroke(color(255));
rectangle.setStrokeWeight(4);
rectangle.setFill(color(127));
}
void draw() {
  background(50);
  //Situamos en el puntero
  translate(mouseX, mouseY);
  shape(rectangle);
}
```

Como se muestra en el listado 2.2, la modificación de características de color de una *PShape* requiere utilizar los métodos *setFill*, *setStroke*, *setStrokeWeight*, etc. En este sentido, el listado 2.3 selecciona el color de relleno de la forma según la posición del puntero.

Listado 2.3: Modificando el color de relleno de una variable *PShape*

```
PShape rectangle;

void setup() {
  size(400,400,P2D);
  rectangle = createShape(RECT,-50,-25,100,50);
  rectangle.setStroke(color(255));
  rectangle.setStrokeWeight(4);
  rectangle.setFill(color(127));
}
void draw() {
  background(50);
  translate(mouseX, mouseY);
  rectangle.setFill(color(map(mouseX, 0, width, 0, 255)));
  shape(rectangle);
}
```

La flexibilidad ofrecida por las variables tipo *PShape* aparece al poder definir los vértices que componen el objeto de forma arbitraria. El tutorial disponible en la web [Processing Foundation](#) [Accedido Enero 2019b] incluye el ejemplo de una estrella (también disponible como ejemplo *Archivo->Ejemplos->Topics->Create Shapes->PolygonPShape*), adaptado en el listado 2.4, y que define los vértices que delimitan a la forma entre llamadas a *beginShape* y *endShape*, esta última con *CLOSE* como argumento para cerrar la línea poligonal.

Listado 2.4: Polígono arbitrario, una estrella, con *PShape*

```
PShape star;

void setup() {
  size(400,400,P2D);

  // La variable
```



```

    star = createShape();
    star.beginShape();
    // El pincel
    star.fill(102);
    star.stroke(255);
    star.strokeWeight(2);
    // Los puntos de la forma
    star.vertex(0, -50);
    star.vertex(14, -20);
    star.vertex(47, -15);
    star.vertex(23, 7);
    star.vertex(29, 40);
    star.vertex(0, 25);
    star.vertex(-29, 40);
    star.vertex(-23, 7);
    star.vertex(-47, -15);
    star.vertex(-14, -20);
    star.endShape(CLOSE);
}

void draw() {
    background(51);
    // Movimiento con el puntero
    translate(mouseX, mouseY);
    // Dibujamos
    shape(star);
}

```

A partir de la forma de estrella, otro ejemplo disponible es *PolygonPShapeOOP2* (*Archivo->Ejemplos->Topics->Create Shapes*). El resultado del código presente en el listado 2.5, hace uso de la clase *Polygon*, ver listado 2.6, mostrando múltiples estrellas en movimiento vertical descendente.

Listado 2.5: Múltiples estrellas

```

ArrayList<Polygon> polygons;

void setup() {
    size(640, 360, P2D);

    // La forma
    PShape star = createShape();
    star.beginShape();
    star.noStroke();
    star.fill(0, 127);
    star.vertex(0, -50);
    star.vertex(14, -20);
    star.vertex(47, -15);
    star.vertex(23, 7);
    star.vertex(29, 40);
    star.vertex(0, 25);
    star.vertex(-29, 40);
}

```

```

star.vertex(-23, 7);
star.vertex(-47, -15);
star.vertex(-14, -20);
star.endShape(CLOSE);

// Lista de objetos
polygons = new ArrayList<Polygon>();

// Lista de objetos PShape
for (int i = 0; i < 25; i++) {
    polygons.add(new Polygon(star));
}
}

void draw() {
    background(255);

    // Dibujamos
    for (Polygon poly : polygons) {
        poly.display();
        poly.move();
    }
}

```

La mencionada clase *Polygon* se encarga de definir la posición inicial y la velocidad de cada forma añadida a la lista de objetos.

Listado 2.6: Clase *Polygon*

```

class Polygon {
    // The PShape object
    PShape s;
    // The location where we will draw the shape
    float x, y;
    // Variable for simple motion
    float speed;

    Polygon(PShape s_) {
        x = random(width);
        y = random(-500, -100);
        s = s_;
        speed = random(2, 6);
    }

    // Simple motion
    void move() {
        y += speed;
        if (y > height+100) {
            y = -100;
        }
    }

    // Draw the object
    void display() {

```

```
pushMatrix();  
translate(x, y);  
shape(s);  
popMatrix();  
}  
}
```

2.2. P3D

Los ejemplos anteriores trabajan en 2D. Processing ofrece diversos modos de reproducción como son: SVG, PDF, P2D y P3D. P2D y P3D en concreto hacen uso de hardware compatible con OpenGL, permitiendo mayor rendimiento. P2D, que ha aparecido en alguno de los listados previos, es el modo de reproducción optimizado para dos dimensiones, mientras que P3D nos permite trabajar en tres dimensiones [Processing Foundation \[Accedido Enero 2019a\]](#). Para ambos modos, la calidad del resultado puede configurarse con los métodos *smooth* y *hint*, si bien lo veremos en prácticas futuras.

Existen formas tridimensionales básicas como la esfera y el prisma, respectivamente con los métodos *sphere* y *box*, ver listado 2.7.

Listado 2.7: Cubo y esfera

```
size(640, 360, P3D);  
background(0);  
  
noFill();  
stroke(255);  
  
// Prisma  
translate(width*0.2, height*0.15, 0);  
box(100);  
  
// Esfera  
translate(width/2, height*0.35, 0);  
sphere(100);
```

Sin embargo, su potencia y flexibilidad, como ya mencionamos, viene dada por poder definir objetos a través de vértices arbitrarios. Como primer ejemplo, el listado 2.8 muestra una pirámide formada por cuatro caras triangulares, cada una con 3 puntos tridimensionales. Al mover el puntero observamos el efecto de la proyección en perspectiva.

Listado 2.8: Pirámide de cuatro lados

```
PShape obj;  
  
void setup() {
```

```
size(600, 600,P3D);

// La variable
obj=createShape();
// El pincel
obj.beginShape();
obj.noFill();
// Puntos de la forma
obj.vertex(-100, -100, -100);
obj.vertex( 100, -100, -100);
obj.vertex( 0, 0, 100);

obj.vertex( 100, -100, -100);
obj.vertex( 100, 100, -100);
obj.vertex( 0, 0, 100);

obj.vertex( 100, 100, -100);
obj.vertex(-100, 100, -100);
obj.vertex( 0, 0, 100);

obj.vertex(-100, 100, -100);
obj.vertex(-100, -100, -100);
obj.vertex( 0, 0, 100);
obj.endShape();
}

void draw() {
  background(255);
  //Movemos con puntero
  translate(mouseX, mouseY);
  //Muestra la forma
  shape(obj);
}
```

La pareja *beginShape-endShape* sin argumentos asume una serie de vértices consecutivos que conforman una línea poligonal. La especificación de un parámetro a la hora de crear la forma permite indicar el tipo de elementos que definen los vértices a continuación: *POINTS*, *LINES*, *TRIANGLES*, *TRIANGLE_FAN*, *TRIANGLE_STRIP*, *QUADS*, o *QUAD_STRIP* (más detalles [aquí¹](https://processing.org/reference/beginShape_.html)):

- *beginShape-endShape(CLOSE)*: Cierra la línea poligonal, uniendo el último vértice con el primero, aplicando el color de relleno.
- *beginShape(POINTS)-endShape*: Cada vértice es un punto, no se conectan con líneas.
- *beginShape(LINES)-endShape*: Cada dos puntos definen un segmento independiente.

¹https://processing.org/reference/beginShape_.html

- *beginShape(TRIANGLES)-endShape*: Cada grupo de tres puntos define un triángulo. Aplica relleno.
- *beginShape(TRIANGLE_STRIP)-endShape*: Los triángulos no son independientes entre sí, cada nuevo vértice compone un triángulo con los últimos dos vértices del triángulo anterior. Aplica relleno.
- *beginShape(TRIANGLE_FAN)-endShape*: El primer vértice se comparte por todos los triángulos. Aplica relleno.
- *beginShape(QUADS)-endShape*: Cada cuatro puntos definen un polígono. Aplica relleno.
- *beginShape(QUAD_STRIP)-endShape*: Se reutilizan los dos últimos vértices del polígono anterior. Aplica relleno.

Como ejemplo de creación de una forma propia, el listado 2.9 muestra una serie enlazada de triángulos creada con la opción *TRIANGLE_STRIP*.

Listado 2.9: Ejemplo de uso de *TRIANGLE_STRIP* con *PShape*

```
PShape obj;

void setup() {
  size(600, 600, P3D);

  // La variable
  obj = createShape();
  // El pincel
  obj.beginShape(TRIANGLE_STRIP);
  obj.fill(102);
  obj.stroke(255);
  obj.strokeWeight(2);
  // Puntos de la forma
  obj.vertex(50, 50, 0);
  obj.vertex(200, 50, 0);
  obj.vertex(50, 150, 0);
  obj.vertex(200, 150, 0);
  obj.vertex(50, 250, 0);
  obj.vertex(200, 250, 0);
  obj.vertex(50, 350, 0);
  obj.vertex(200, 350, 0);
  obj.vertex(50, 450, 0);
  obj.vertex(200, 450, 0);
  obj.endShape();
}

void draw() {
  background(255);
}
```

```
//Movemos con puntero
translate(mouseX, mouseY);
//Muestra la forma
shape(obj);
}
```

La variante mostrada en el listado 2.10 modifica el valor de la coordenada z de varios vértices. Observa el efecto de la perspectiva sobre los vértices más alejados.

Listado 2.10: Ejemplo de uso de *TRIANGLE_STRIP* con profundidad

```
PShape obj;

void setup() {
  size(600, 600, P3D);

  // La variable
  obj = createShape();
  // El pincel
  obj.beginShape(TRIANGLE_STRIP);
  obj.fill(102);
  obj.stroke(255);
  obj.strokeWeight(2);
  // Puntos de la forma
  obj.vertex(50, 50, 0);
  obj.vertex(200, 50, 0);
  obj.vertex(50, 150, 0);
  obj.vertex(200, 150, -100);
  obj.vertex(50, 250, -100);
  obj.vertex(200, 250, -100);
  obj.vertex(50, 350, -200);
  obj.vertex(200, 350, -200);
  obj.vertex(50, 450, -200);
  obj.vertex(200, 450, -200);
  obj.endShape();
}

void draw() {
  background(255);
  //Movemos con puntero
  translate(mouseX, mouseY);
  //Muestra la forma
  shape(obj);
}
```

2.3. SÓLIDO DE REVOLUCIÓN

La creación de objetos 3D resulta engorrosa al ser necesario disponer de mecanismos para definir los vértices que delimitan el objeto en un escenario tridimensional, y esto debe

hacerse sobre una pantalla bidimensional. Una posible simplificación del proceso viene dada a través de la creación de un objeto por medio de superficies de barrido o revolución. En ambos casos, se definen en primer lugar una serie de puntos, que conforman una curva plana (generalmente), que bien por sucesivas traslaciones (barrido) o rotaciones (revolución) permiten definir la malla de un objeto tridimensional. De modo ilustrativo, la figura 2.1 crea dos objetos utilizando el esquema de revolución, al definir sendos perfiles que se rotan un número determinado de veces, en este caso sobre el eje y , para crear el objeto a partir de la unión de los sucesivos *meridianos* que conforman la malla del objeto.

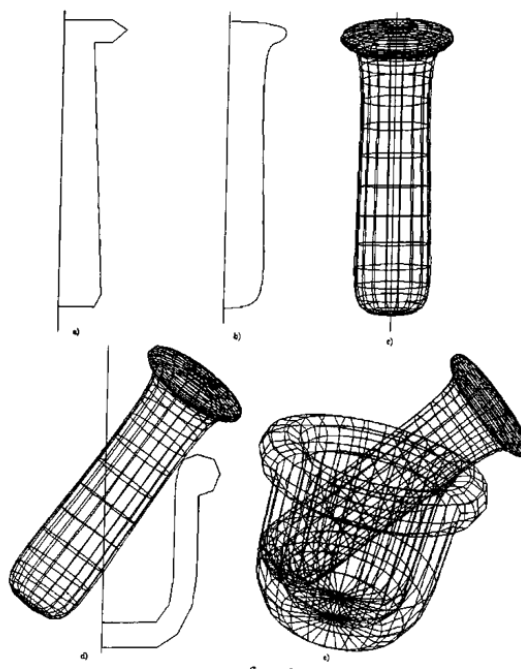


Figura 2.1: Creando una escena con superficies de revolución

Si presentamos la malla resultante *planchada* sobre un plano, tendría el aspecto de una rejilla rectangular, ver figura 2.2 izquierda, donde cada polígono o cara está delimitado por cuatro vértices. Habitualmente resulta más interesante trabajar con triángulos, que pueden crearse de forma sistemática subdividiendo cada polígono, ver figura 2.2 derecha.

2.3.1. Rotación de un punto 3D

En dos dimensiones la rotación de un punto sobre el plano cartesiano se ilustra en la figura 2.3. Siguiendo la regla de la mano derecha, al rotar un ángulo θ el punto p con

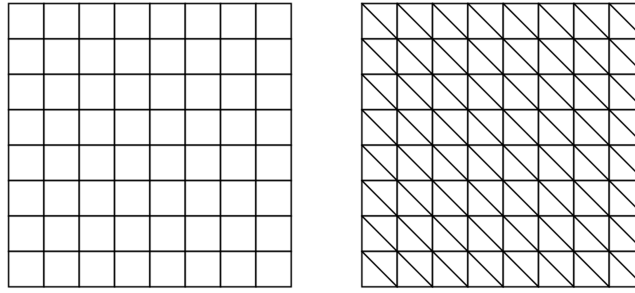


Figura 2.2: Ilustración de malla antes de tras triangularizar

coordenadas (x_1, y_1) , las coordenadas resultantes (x_2, y_2) tras la rotación

$$\begin{aligned} x_2 &= x_1 \cdot \cos(\theta) - y_1 \cdot \sin(\theta) \\ y_2 &= x_1 \cdot \sin(\theta) + y_1 \cdot \cos(\theta) \end{aligned} \quad (2.1)$$

O su equivalente en forma matricial con premultiplicación:

$$(x_2, y_2) = (x_1, y_1) \cdot \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{bmatrix} \quad (2.2)$$

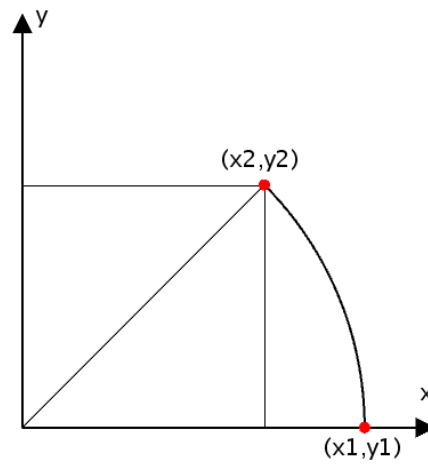


Figura 2.3: Rotación 2D de un ángulo θ

Extensible de forma análoga a tres dimensiones, donde por simplicidad asumiremos una rotación de un punto alrededor del eje vertical y , de forma similar al ejemplo mostrado en la figura 2.1. De esta forma, las coordenadas rotadas de un punto 3D rotado un ángulo θ sobre el eje y siguen las siguientes expresiones:

$$x_2 = x_1 \cdot \cos(\theta) - z_1 \cdot \sin(\theta)$$

$$y_2 = y_1$$

$$z_2 = x_1 \cdot \sin(\theta) + z_1 \cdot \cos(\theta)$$

(2.3)

$$(x_2, y_2, z_2) = (x_1, y_1, z_1) \cdot \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{bmatrix}$$

Un sólido de revolución requiere la rotación un número de veces de los puntos del perfil proporcionado para su obtención.

2.4. TAREA

Crear un prototipo que recoja puntos de un perfil al hacer clic con el ratón sobre la pantalla, para finalmente crear un objeto tridimensional por medio de una superficie de revolución, almacenando la geometría resultante en una variable de tipo *PShape*, ver a modo de ilustración la figura 2.4. El prototipo permitirá crear sólidos de revolución de forma sucesiva, si bien se almacene únicamente el último definido.

La entrega se debe realizar a través del campus virtual, remitiendo una memoria o el enlace a un screencast, que además de describir las decisiones adoptadas, identifique al remitente, además de las referencias y herramientas utilizadas, cuidando el formato y estilo, debe incluir enlace al código desarrollado (p.e. github).

BIBLIOGRAFÍA

Processing Foundation. P3D tutorial, Accedido Enero 2019a. URL <https://processing.org/tutorials/p3d/>.

Processing Foundation. PShape tutorial, Accedido Enero 2019b. URL <https://processing.org/tutorials/pshape/>.

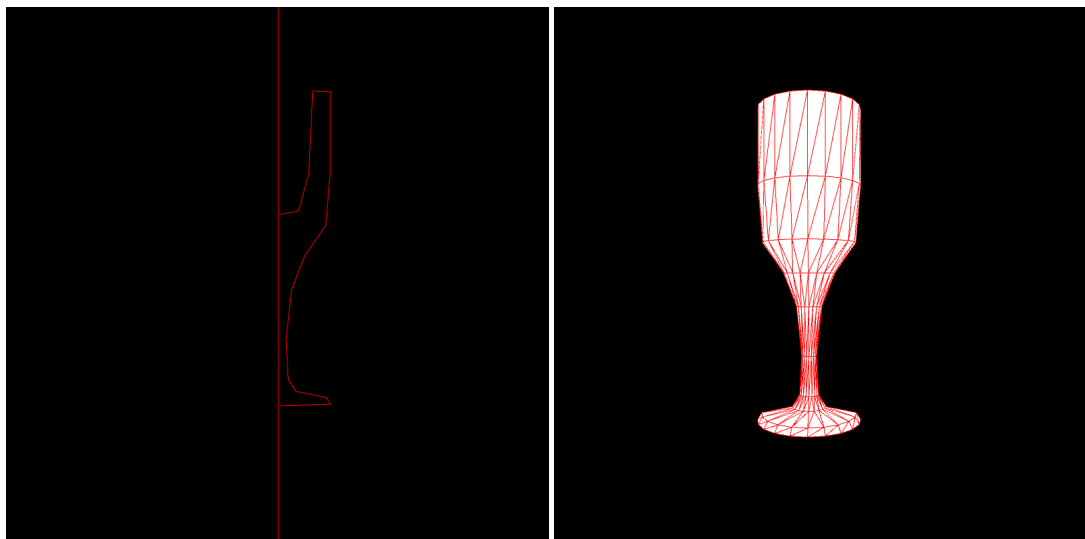


Figura 2.4: Perfil y sólido de revolución resultante